プログラミング言語を作ろう

～代数データ型によるプログラミング言語の表現と処理編～

A:Mac合宿 2022/2/19-20

プログラミング言語を作ろう

この発表では、Haskellを使って、標準ライブラリ以外のライブラリを一切使わずに簡単なインタプリタ言語を実装する。

- 前半は、プログラミング言語を表現するデータ構造を構築し、そのデータを実行するプログラムを実装する
- 後半は、テキストで書かれたソースコードから前半で構築したデータ構造への変換機能を実装する

このスライドとソースコード

GitHubの以下のリポジトリからダウンロードできる。

https://github.com/elkel53930/amac2022_general_session

```
> git clone git@github.com:elkel53930/amac2022_general_session.git
```

今日限定でpublicにしておくので、ダウンロードしたい方は今日中に。

Haskellにおける「=」の意味

- Haskellを使って説明する。
- Haskellでは「=」演算子は「定義」を意味する、と覚えておくとよい。
  「A = B」はAにBを代入するという意味ではなく、AはBであるという意味。
- 還元すれば、「AはBに置き換えることができる」ということ。
- 例えば

```
f x y = x + y
```

と定義すれば、`f 1 2`は`1 + 2`に置き換えることができる。
さらに言えば、置き換えられる限り置き換え続ける。

代数データ型によるプログラミング言語の表現と処理

- 最近の言語の多く(Rust, Swift, F#, Haskell, OCamlなど)に実装されている機能で、様々なデータ構造を表現できる。
- C言語が`struct`や`enum`や`union`でデータ構造を構築するように、これらの言語では代数データ型を用いてデータ構造を構築する。
- 今回は
  - 代数データ型を用いてプログラミング言語を表すデータ構造を構築し、
  - さらにそのデータを実行する(インタプリタとして動作する)コードを実装する

Haskellのインストール

かんたん

```
> sudo apt install haskell-platform
```

このスライドで使用するHaskellの機能

- 代数データ型
- 再帰的な型
- 関数
- パターンマッチ
- 再帰的な関数
- リスト
- タプル

型(1)

型とはなにか。

- Cのcharは-128〜127の値をとりうる。
- Cのintは-2,147,483,648〜2,147,483,647の値をとりうる。
- C++のstringはすべての有限長の文字列をとりうる。
- (正確には違うけど、この発表の中では)型とは取りうる値の集合といえる。
- 型を指定することで、とりうる値を集合の中に制限することができる。

型(2)

- 「曜日」をあらわす変数をchar型に指定することを考える。
  - 日曜日:0, 月曜日:1 ... 土曜日:6
- では-127〜-1と7〜128は？
- 表現したいものに対して変数の型が「大きすぎる」とバグのもと。

直和型(1)

```
data DayOfWeek = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving Show
```

Cで書くなら

```
enum {
    Sun = 0,
    Mon, Tue, Wed,
    Thu, Fri, Sat,
}DayOfWeek;
```

直和型(2)

```
data DayOfWeek = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving Show
```

- `Sun` や `Mon` といった“値”を**コンストラクタ**という。
- 型全体は、それぞれの要素の**和集合**になっているので直和型という。
- `deriving Show` は、文字列に変換できるようになるおまじない。

直積型(1)

```
data Person = Person String Int -- Name and Age
```

Cで書くなら

```
struct Person {
    char* name;
    int age;
};
```

直積型(2)

```
data Person = Person String Int -- Name and Age
```

- `Person`はコンストラクタ。
- 型全体は、`Int`という集合と`String`という集合の**積集合**になっているので、直積型という。

代数データ型

- 直和型と直積型と合わせて**代数データ型**という。
- 直和型と直積型は「まぜて」使用することができる。

```
data Shape = Circle Double -- Diameter
           | Rectangle Double Double -- Width, Height
           | Line Double Double -- Angle, Length
```

Cで無理やり書くと

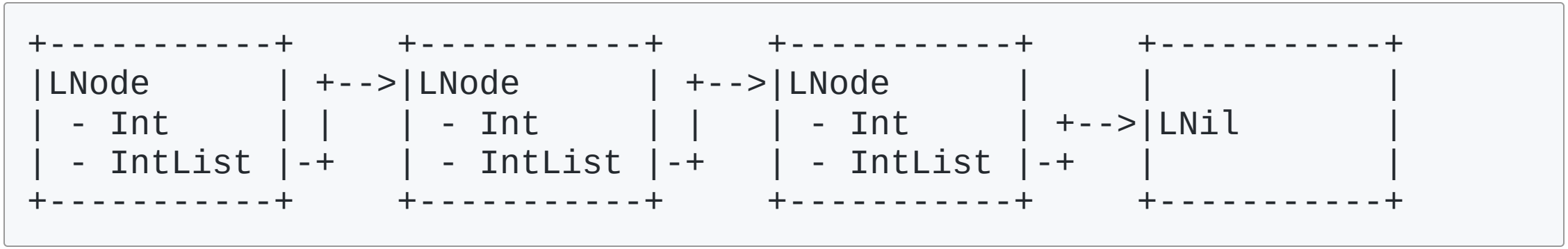
```
union Shape {
    struct { double diameter; }circle;
    struct { double width; double height; }rectangle;
    struct { double angle; double length; }line;
};
```

（Cの場合は、代入されている値がどの構造体なのかの情報が更に必要）

再帰的な型(1)

```
data IntList = LNode Int IntList
             | LNil
```

- IntListはNodeもしくはNilである。
- NodeはIntとIntList、２つの値を持っている。
- Nilは値を持っていない。



- 整数値のリスト構造ができる。

再帰的な型(2)

- 木構造

```
data IntTree = TNode Int IntTree IntTree
             | TNil
```

- 四則演算からなる式（計算の順序も表現されていることに注意）

```
data Op = Sum | Sub | Mul | Div
data Expr = Literal Int
          | BiOp Op Expr Expr
```

たとえば`(1 + 2) * (6 / 3)`は

```
BiOp Mul
     (BiOp Sum (Literal 1) (Literal 2))
     (BiOp Div (Literal 6) (Literal 3))
```

関数

- Haskellでは関数は以下のような構文で定義する。

```
関数名 :: 引数１の型 -> 引数２の型 ... -> 戻り値の型
関数名 = 関数の定義
```

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

```haskell
cond :: Bool -> Int -> Int -> Int
cond condition x y =
    if condition == True then
        x
    else
        y
```

パターンマッチ(1)

```
data DayOfWeek = Sun | Mon | Tue | Wed | Thu | Fri | Sat

isHoliday :: DayOfWeek -> Bool
isHoliday Sun = True
isHoliday Mon = False
isHoliday Tue = False
isHoliday Wed = False
isHoliday Thu = False
isHoliday Fri = False
isHoliday Sat = True
```

- 上から順にマッチするパターンを探していく。

パターンマッチ(2)

- 変数にマッチさせることもできる。

```
data DayOfWeek = Sun | Mon | Tue | Wed | Thu | Fri | Sat

isHoliday' :: DayOfWeek -> Bool
isHoliday' Sun = True
isHoliday' Sat = True
isHoliday' d = False
```

- でもこの変数はつかわれていない…

パターンマッチ(3)

- `_` は「使われない変数」を表す。

```
data DayOfWeek = Sun | Mon | Tue | Wed | Thu | Fri | Sat

isHoliday'' :: DayOfWeek -> Bool
isHoliday'' Sun = True
isHoliday'' Sat = True
isHoliday'' _ = False
```

パターンマッチ(4)

- それぞれのコンストラクタが持つ値を取り出すこともできる。

```
data Shape = Circle Double -- Diameter
           | Rectangle Double Double -- Width, Height
           | Line Double Double -- Angle, Length

area :: Shape -> Double

area (Circle diameter) = radius * radius * 3.14
    where radius = diameter / 2

area (Rectangle width height) = width * height

area (Line _ _) = 0
```

再帰的な代数データ型と再帰関数

- IntList内の値を全部足し合わせたい

```
data IntList = LNode Int IntList
             | LNil

total :: IntList -> Int
-- 要素のないリストの合計値はゼロ
total LNil = 0
-- あるリストの合計値は、先頭の値 + 残りのリストの合計値
total (LNode value next) = value + (total next)
```

再帰的な代数データ型と再帰関数

```
data IntList = LNode Int IntList
             | LNil

total :: IntList -> Int
-- 要素のないリストの合計値はゼロ
total LNil = 0
-- あるリストの合計値は、先頭の値 + 残りのリストの合計値
total (LNode value next) = value + (total next)
```

「=の左辺は右辺で置き換えられる」ルールを思い出そう。

```
total (LNode 10 (LNode 9 (LNode 8 LNil)))
10 + total (LNode 9 (LNode 8 LNil))
10 + 9 + total (LNode 8 LNil)
10 + 9 + 8 + total LNil
10 + 9 + 8 + 0
27
```

リスト(1)

実はリストは最初からHaskellに用意されている

```
> [0,1,2,3,4,5]
[0,1,2,3,4,5]
> [0..5]
[0,1,2,3,4,5]
> let empty_list = []
> empty_list
[]
```

`Nil`に相当するのは`[]`。
リストのパターンマッチは

```
total' :: [Int] -> Int
total' [] = 0
total' (x:xs) = x + total' xs
```

`x`が先頭の要素と、`xs`がそれ以降の要素とマッチする。

リスト(2)

```
total' :: [Int] -> Int
total' [] = 0
total' (x:xs) = x + total' xs
```

「=の左辺は右辺で置き換えられる」ルールを思い出そう。

```
total' [1,2,3]
1 + total' [2,3]
1 + 2 + total' [3]
1 + 2 + 3 + total' []
1 + 2 + 3 + 0
6
```

リスト(3)

リストの結合は`++`演算子で行う。

```
> [0,1,2,3] ++ [4,5,6,7]
[0,1,2,3,4,5,6,7]
```

タプル

タプルを使うと、複数の異なる型の要素を格納することができる。

```
(1,"Number") -- IntとStringのタプル

(fromGregorian 2022 2 20, Sun) -- DayとDayOfWeekのタプル

(1, 2, 3.0) -- IntとIntとDoubleのタプル
```

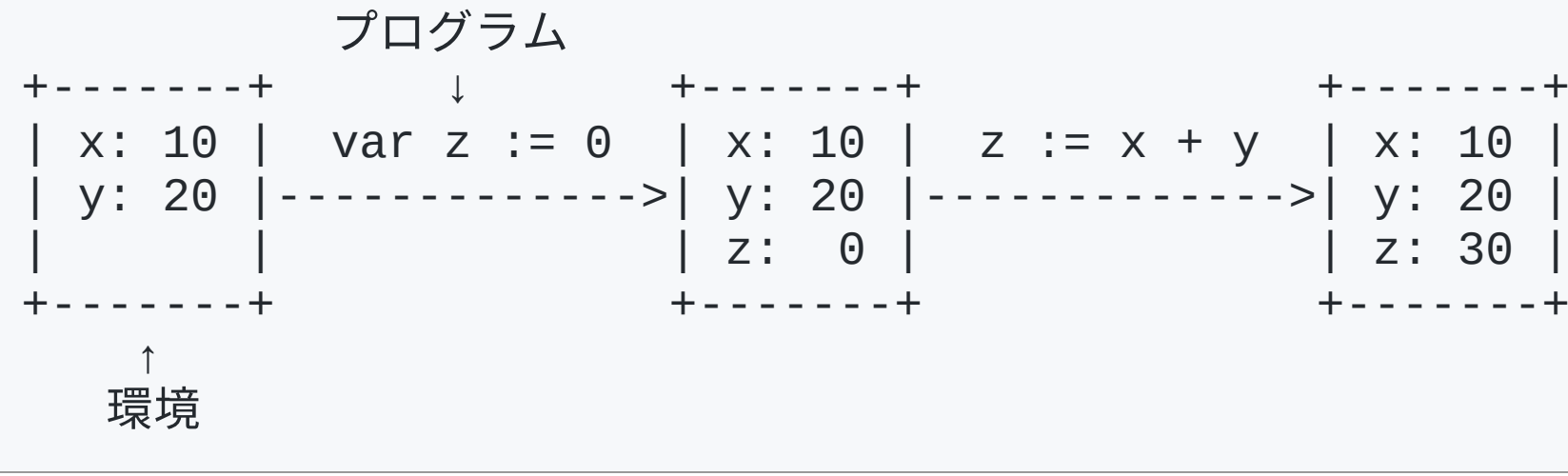
実装方針

環境(environment)

- 「変数名」と「変数の値」のリストを「環境(environment)」と呼ぶことにする。
- 環境は`Environment`という型で表現され、プログラムを実行するたびに環境が更新されていく。

```
type VariableName = String
type BaseType = Int

type Environment = [(VariableName, BaseType)] -- 変数名と値のタプルのリスト
```



式(expression)

値は式というデータ構造によって表現される。

```
data BinaryOperator
    = Add | Sub | Mul | Div -- 四則演算子
    | Lt | Le | Gt | Ge | Eq | Neq -- 比較演算子
    | And | Or -- 論理演算子

data Expression
    = Literal BaseType -- 整数リテラル
    | Variable VariableName -- 変数
    | BinaryOperation BinaryOperator Expression Expression -- 二項演算
```

例えば `((1 + 2) <= x)` は

```
BinaryOperation Le (BinaryOperation Add (Literal 1) (Literal 2)) (Variable "x")
```

式の評価: 整数リテラル

評価 = 結果を得ること。式を評価すると整数が得られる。
整数リテラルを評価するにはその整数を返せば良い。

```
expressionEvaluation environment (Literal value) = value
```

動作確認

```
> expressionEvaluation [] (Literal 10)
10
```

式の評価: 変数

変数を評価するには環境から変数名を探し出し、その値を返せば良い。

```
getValue :: Environment -> VariableName -> BaseType
getValue [] _ = undefined
getValue ((name, value) : next) variable_name =
    if variable_name == name then
        value
    else
        getValue next variable_name

. . .

expressionEvaluation environment (Variable name) = getValue environment name
```

動作確認

```
> expressionEvaluation [("x",123),("y",456)] (Variable "y")
456
```

式の評価: 二項演算

二項演算を評価するには

```
expressionEvaluation environment (BinaryOperation operator x y) =
    binaryOperatorEvaluation operator x' y'
    where
        x' = expressionEvaluation environment x
        y' = expressionEvaluation environment y
```

左辺の式を右辺の式を評価し、その結果同士を演算して返せば良い。

`expressionEvaluation`を再帰的に呼び出しており、二項演算の左辺も右辺も、整数リテラル、変数、二項演算のどれにでもなりうる。

式の評価: 二項演算

```
toBool :: Bool -> BaseType
toBool True = 1
toBool False = 0

fromBool :: BaseType -> Bool
fromBool 0 = False
fromBool _ = True

binaryOperatorEvaluation :: BinaryOperator -> BaseType -> BaseType -> BaseType
binaryOperatorEvaluation Add x y = x + y
binaryOperatorEvaluation Sub x y = x - y
binaryOperatorEvaluation Mul x y = x * y
binaryOperatorEvaluation Div x y = x `div` y
binaryOperatorEvaluation Lt x y = toBool (x < y)
binaryOperatorEvaluation Le x y = toBool (x <= y)
binaryOperatorEvaluation Gt x y = toBool (x > y)
binaryOperatorEvaluation Ge x y = toBool (x >= y)
binaryOperatorEvaluation Eq x y = toBool (x == y)
binaryOperatorEvaluation Neq x y = toBool (x /= y)
binaryOperatorEvaluation And x y = toBool (fromBool x && fromBool y)
binaryOperatorEvaluation Or x y = toBool (fromBool x || fromBool y)
```

式の評価

試しに`(10 - (1 + 2))`を評価してみる。

```
> e = BinaryOperation Sub (Literal 10) (BinaryOperation Add (Literal 1) (Literal 2))
> expressionEvaluation [] e
7
```

変数は使っていないので、カラの環境 = 変数の何も登録されていない環境`[]`を渡している。

文(Statement)

実行されるプログラムは文というデータ構造で表現される。

```
data Statement
    = DeclareVariable VariableName Expression -- 変数宣言
    | Sequence Statement Statement -- 複数行のプログラム
    | Assign VariableName Expression -- 代入
    | If Expression Statement Statement -- if文
    | Pass -- なにもしない
    | While Expression Statement -- while文
```

- `Statement`を実行する関数は

```
statementEvaluation :: Environment -> Statement -> Environment
```

- 「実行前の環境」と「プログラム」を渡すと「実行後の環境」を返す関数

変数宣言を表すデータ構造

- 変数宣言は「変数名」と「初期値」の２つの要素からなる。
- イメージとしては「var **xxx** := **10**;」。

```
data Statement = DeclareVariable VariableName Expression
--                               変数名       初期値
```

文の評価: 変数宣言

変数宣言を評価するには、与えられた環境にその変数名と初期値の評価結果を追加して返せば良い。

```
statementEvaluation environment (DeclareVariable variable_name initial_value)
    = [(variable_name, initial_value')] ++ environment
        where initial_value' = expressionEvaluation environment initial_value
```

動作確認

```
> statementEvaluation [] (DeclareVariable "x" (BinaryOperation Add (Literal 1) (Literal 2)))
[("x",3)]
```

複数行のプログラムを表すデータ構造

```
data Statement
    (中略)
    | Sequence Statement Statement -- 複数行のプログラム
--                  ↑           ↑
--             何かしらの文 次のSequence
```

例えば、`x`, `y`, `z` ３つの変数を宣言する場合は

```
Sequence
    (DeclareVariable "x" (Literal 0))
    (Sequence
        (DeclareVariable "y" (Literal 0))
        (DeclareVariable "z" (Literal 0)))
```

文の評価: 複数行のプログラム

１つ目のステートメントの評価結果(評価後の環境)を使って２つ目のステートメントを評価し、その結果を返せば良い。

```
statementEvaluation environment (Sequence statement1 statement2)
    = statementEvaluation environment' statement2
        where environment' = statementEvaluation environment statement1
```

動作確認

```
> s = (中略、前の例と同じ)
> statementEvaluation [] s
[("z",0),("y",0),("x",0)]
```

代入を表すデータ構造

```
data Statement
    (中略)
    | Assign VariableName Expression -- 代入
--                   ↑           ↑
--            代入先の変数 代入する値の式
```

例えば、変数`a`に`(1 + a)`を代入する場合は

```
Assign "a" (BinaryOperation Add (Literal 1) (Variable "a"))
```

文の評価: 代入

代入は環境を変化させるので、まず環境内にある変数の値を更新する関数を作る。

```
updateEnvironment :: Environment -> VariableName -> BaseType -> Environment
updateEnvironment [] _ _ = undefined
updateEnvironment ((name, current_value) : next) variable_name value =
    if variable_name == name then -- 名前が一致すれば、
        [(name, value)] ++ next -- 新しい値をセットして環境の「残り」を連結して返す
    else
        -- 名前が一致しなければ、環境の「残り」を引数にして自分自身を再帰的に呼び出す
        [(name, current_value)] ++ (updateEnvironment next variable_name value)
```

例えば`updateEnvironment [("x",0), ("y",1), ("z",2)] "y" 5`を評価すると

```
updateEnvironment [("x",0), ("y",1), ("z",2)] "y" 5
-- name:"x", current_value:0, next:[("y",1), ("z",2)], variable_name:"y", value:5
[("x",0)] ++ updateEnvironment [("y",1), ("z",2)] "y" 5
-- name:"y", current_value:1, next:[("z",2)], variable_name:"y", value:5
[("x",0)] ++ [("y",5)] ++ [("z",2)]
[("x",0), ("y",5), ("z",2)]
```

文の評価: 代入

式を評価し、その値で変数を(環境を)更新すれば良い。

```
statementEvaluation environment (Assign variable_name value)
    = updateEnvironment environment variable_name value'
        where value' = expressionEvaluation environment value
```

動作確認

```
> statementEvaluation [("a",100)] (Assign "a" (Literal 10))
[("a",10)]
```

if文を表すデータ構造

```
data Statement
    (中略)
    | If Expression Statement Statement
--              ↑           ↑           ↑
--          条件式   真の時の文     偽の時の文
```

例えば、変数 `a` が `1` ならプラス１、そうでなければマイナス１する場合は

```
If (BinaryOperation Eq (Variable "a") (Literal 1))
    (Assign "a" (BinaryOperation Add (Variable "a") (Literal 1)))
    (Assign "a" (BinaryOperation Sub (Variable "a") (Literal 1)))
```

文の評価: if文

条件式(`condition`)を評価し、その結果が0なら`else_statement`の評価結果を、0でなければ`then_statement`の評価結果を返す。

```
statementEvaluation environment (If condition then_statement else_statement)
    = if expressionEvaluation environment condition == 0 then
          statementEvaluation environment else_statement
      else
          statementEvaluation environment then_statement
```

動作確認

```
> s = If (BinaryOperation Eq (Variable "a") (Literal 1))
    (Assign "a" (BinaryOperation Add (Variable "a") (Literal 1)))
    (Assign "a" (BinaryOperation Sub (Variable "a") (Literal 1)))
> statementEvaluation [("a",1)] s
[("a",2)]
> statementEvaluation [("a",10)] s
[("a",9)]
```

「なにもしない」を表すデータ構造とその評価

今回のif文には必ずelseが必要なので、なにもしない文`pass`を用意する。

```
data Statement
    (中略)
    | Pass
```

これを評価しても、実行前の環境をそのまま返すだけ。

```
statementEvaluation environment Pass = environment
```

whileを表すデータ構造

```
data Statement
    (中略)
    | While Expression Statement
--                  ↑           ↑
--              条件式       繰り返す文
```

たとえば `a` が `10` より大きくなるまで５を足し続ける場合は

```
While (BinaryOperation Le (Variable "a") (Literal 10))
    (Assign "a" (BinaryOperation Add (Variable "a") (Literal 5)))
```

文の評価: while文

条件式(`condition`)を評価し、その結果が0なら渡された環境をそのまま返し、0でないなら、`statement`を評価したあとの環境(`environment'`)でもう一度`While`を作って自分自身(`statementEvaluation`)に渡す。

```
statementEvaluation environment (While condition statement)
    = if expressionEvaluation environment condition == 0 then
          environment
      else
          statementEvaluation environment' (While condition statement)
              where environment' = statementEvaluation environment statement
```

動作確認

```
> s = While (BinaryOperation Le (Variable "a") (Literal 10))
    (Assign "a" (BinaryOperation Add (Variable "a") (Literal 5)))
> statementEvaluation [("a",2)] s
[("a",12)]
> statementEvaluation [("a",20)] s
[("a",20)]
```

式と文のまとめ

式

- 整数リテラル、変数、二項演算からなる。
- 二項演算は引数に式を二つとる。ネストが可能。

文

- 変数宣言
- 複数行の文
- 代入
- if文
- なにもしない文(pass)
- while文

代数データ型でプログラムを書いてみる

さて、これは何をするプログラムでしょう？

```
 Sequence (DeclareVariable "x" (Literal 8))
(Sequence (DeclareVariable "result" (Literal 1))
          (While (BinaryOperation Neq (Variable "x") (Literal 0))
              (Sequence (Assign "result"
                                (BinaryOperation Mul (Variable "result") (Variable "x")))
                        (Assign "x"
                                (BinaryOperation Sub (Variable "x") (Literal 1))))))
```

代数データ型でプログラムを書いてみる

正解は`8!`つまり、8の階乗を計算するプログラム。

```
> s = (中略)]
> statementEvaluation [] s
[("result",40320),("x",0)]
```

`8! = 40320`なので、ちゃんと変数`result`に8の階乗が代入されている。

代数データ型ではわかりにくい

後半では、テキストで書かれた文字列をデータ構造に変換するパーサを実装する。

`8!` の例は、次のようになる

```
var x:= 8;
var result := 1;
while (x != 0) {
    result := (result * x);
    x := (x - 1)
}
```